Data Structures — Solutions

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti
Spelman College

Problem 1: Choosing the Right Structure

(a)

Solution

Stack. Each page visited is pushed onto the stack. Pressing back pops the most recently visited page. The LIFO property matches exactly the behavior needed — the last page you visited is the first one you return to.

(b)

Solution

Min-heap (priority queue). The dominant operation is always retrieving the most critical patient next. A min-heap supports extract-min in $O(\log n)$ and insert in $O(\log n)$, which is optimal for a dynamic collection where the minimum must be repeatedly found and removed.

(c)

Solution

Hash table. The dominant operation is lookup by word. A hash table gives $O(1)$ average search, far faster than the $O(\log n)$ of a BST or $O(n)$ of a list. Since the dictionary never changes, there are no insert or delete concerns, and the hash table can be built once at startup.

(d)

Solution

Deque (double-ended queue). Songs must be added or removed from either end in $O(1)$. A deque supports `pushFront`, `pushBack`, `popFront`, and `popBack` all in $O(1)$.

(e)

Solution

Stack. Scan left to right. When an opening bracket is seen, push it. When a closing bracket is seen, check whether it matches the top of the stack (the most recently opened unmatched bracket). If it matches, pop; otherwise the string is invalid. LIFO order is exactly what is needed — the most recently opened bracket must be closed first.

Problem 2: Tree Traversals

(a)

Solution

Inorder: 10, 20, 30, 40, 50, 60, 70, 80, 90.
The result is sorted in ascending order. This is always the case for inorder traversal of a BST — the BST property guarantees that every left subtree contains smaller values and every right subtree contains larger values, so visiting left-root-right yields sorted output.

(b)

Solution

Preorder: 50, 30, 20, 10, 40, 70, 60, 80, 90.

(c)

Solution

Postorder: 10, 20, 40, 30, 60, 90, 80, 70, 50.

(d)

Solution

Level-order: 50, 30, 70, 20, 40, 60, 80, 10, 90.

(e)

Solution

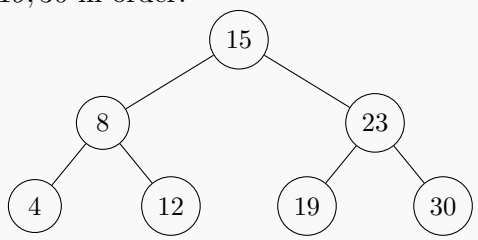
Preorder. To copy a tree, you must create a node before you can attach its children. Preorder visits the root first, then the left subtree, then the right subtree — so you create each node exactly when you need it, before recursing into its children. Inorder or postorder would require you to know the children before creating the parent, which is impossible.

Problem 3: BST Operations

(a)

Solution

Inserting $15, 8, 23, 4, 12, 19, 30$ in order:



(b)

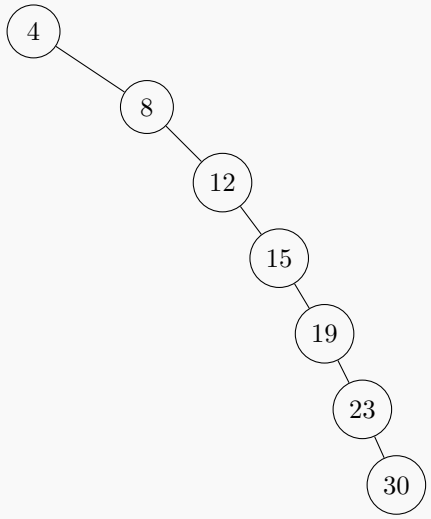
Solution

Height = 2. The longest root-to-leaf path is $15 \to 8 \to 4$ (or any other leaf), which has 2 edges.

(c)

Solution

Inserting $4, 8, 12, 15, 19, 23, 30$ in sorted order produces a degenerate (right-skewed) tree:



Height = 6.

(d)

Solution

Inserting in sorted order causes every new node to go to the right of the previous one, producing a linked list rather than a balanced tree. The height grows to $n - 1$ instead of $\log n$.

For tree (a): worst-case search is $O(\log n)$ — specifically $O(2) = O(1)$ for this 7-node tree.
For tree (c): worst-case search is $O(n)$ — specifically $O(6)$ for this 7-node tree, since you may have to traverse the entire chain.

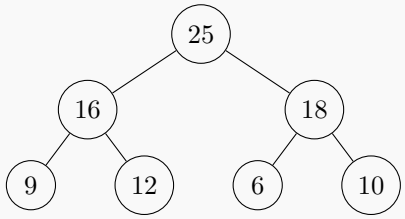
This illustrates why insertion order matters and why self-balancing trees (AVL, Red-Black) are needed in practice.

Problem 4: Heaps

Array: `[25, 16, 18, 9, 12, 6, 10]`

Solution

(a)



Not a valid min-heap. Violations:

- Index 0 (value 25): children are 16 and 18. $25 > 16$ — violation.
- Index 1 (value 16): children are 9 and 12. $16 > 9$ — violation.
- Index 2 (value 18): children are 6 and 10. $18 > 6$ — violation.

Solution

Last non-leaf $= \lfloor 7/2 \rfloor - 1 = 2$ (value 18). Work right-to-left:

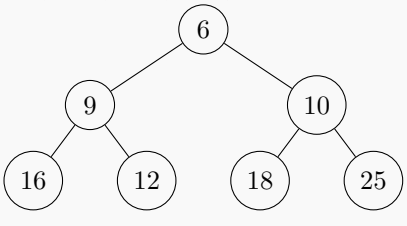
$i = 2$**, value 18:** children are index 5 (value 6) and index 6 (value 10). Smaller child is 6. $18 > 6$, swap.
`[25, 16, 6, 9, 12, 18, 10]`

$i = 1$**, value 16:** children are index 3 (value 9) and index 4 (value 12). Smaller child is 9. $16 > 9$, swap.
`[25, 9, 6, 16, 12, 18, 10]`
Continue bubbling 16 down: index 3 has children at index 7 and 8 — none exist. Stop.

$i = 0$**, value 25:** children are index 1 (value 9) and index 2 (value 6). Smaller child is 6. $25 > 6$, swap.
`[6, 9, 25, 16, 12, 18, 10]`
Continue bubbling 25 down from index 2: children are index 5 (value 18) and index 6 (value 10). Smaller child is 10. $25 > 10$, swap.
`[6, 9, 10, 16, 12, 18, 25]`
Index 6 is a leaf. Stop.

Final min-heap: `[6, 9, 10, 16, 12, 18, 25]`

(b)



(c)

Solution

Insert 5: append to end.
`[6, 9, 10, 16, 12, 18, 25, 5]` (index 7)

Bubble up: parent of 7 is index 3 (value 16). $5 < 16$, swap.
`[6, 9, 10, 5, 12, 18, 25, 16]`
Parent of 3 is index 1 (value 9). $5 < 9$, swap.
`[6, 5, 10, 9, 12, 18, 25, 16]`
Parent of 1 is index 0 (value 6). $5 < 6$, swap.
`[5, 6, 10, 9, 12, 18, 25, 16]`
Root reached. **Final heap:** `[5, 6, 10, 9, 12, 18, 25, 16]`

(d)

Solution

Extract min (value 5). Move last element (16) to root, reduce size.
`[16, 6, 10, 9, 12, 18, 25]`
Bubble down 16 from index 0: children are index 1 (value 6) and index 2 (value 10). Smaller child is 6. $16 > 6$, swap.
`[6, 16, 10, 9, 12, 18, 25]`
Continue from index 1: children are index 3 (value 9) and index 4 (value 12). Smaller child is 9. $16 > 9$, swap.
`[6, 9, 10, 16, 12, 18, 25]`
Index 3 has no children (indices 7 and 8 out of range). Stop.
Final heap: `[6, 9, 10, 16, 12, 18, 25]`

Problem 5: Hash Tables

$m = 7$, $h(k) = k \bmod 7$.

(a)

Solution

First compute the slot for each key:

Key	$k \bmod 7$	Slot
23	$23 \bmod 7 = 2$	2
11	$11 \bmod 7 = 4$	4
30	$30 \bmod 7 = 2$	2 (collision with 23)
4	$4 \bmod 7 = 4$	4 (collision with 11)
17	$17 \bmod 7 = 3$	3
38	$38 \bmod 7 = 3$	3 (collision with 17)

Chaining table:

Slot	Chain
0	(empty)
1	(empty)
2	$23 \to 30$
3	$17 \to 38$
4	$11 \to 4$
5	(empty)
6	(empty)

(b)

Solution

$\alpha = n/m = 6/7 \approx 0.857$.
This is above the recommended threshold of 0.7, meaning the table is fairly full. Expected search time with chaining is $O(1 + \alpha) = O(1.857)$, so on average we expect to traverse about 1–2 nodes per lookup. Performance is beginning to degrade.

(c)

Solution

Linear probing — probe $h(k), h(k)+1, \ldots \pmod 7$ until an empty slot is found:

Key	Natural slot	Probing	Final slot
23	2	—	2
11	4	—	4
30	2	2 full → try 3	3
4	4	4 full → try 5	5
17	3	3 full → try 4, 4 full → try 5, 5 full → try 6	6
38	3	3,4,5,6 full → try 0	0

Final table: slot 0 → 38, slot 2 → 23, slot 3 → 30, slot 4 → 11, slot 5 → 4, slot 6 → 17, slot 1 empty.

(d)

Solution

Natural slots: Both methods placed the same 3 keys (23, 11) directly — keys that did not collide. With chaining, every key lands in its natural slot (collisions are handled by extending the chain). With linear probing, only keys that arrive first at their slot land there; later collisions get displaced.

Performance at this load factor: Chaining is likely to perform better here. At $\alpha \approx 0.86$, linear probing suffers from *primary clustering* — displaced keys block subsequent keys, creating long runs of filled slots (as seen with key 17 needing 4 probes and key 38 needing 5). Chaining degrades more gracefully: the expected chain length is $\alpha \approx 0.86$, so lookups are still close to $O(1)$.

Problem 6: Stacks and Queues

(a)

Solution

Operation	Stack (top on left)
`push(3)`	[3]
`push(7)`	[7, 3]
`push(1)`	[1, 7, 3]
`pop()` returns 1	[7, 3]
`push(5)`	[5, 7, 3]
`push(2)`	[2, 5, 7, 3]
`pop()` returns 2	[5, 7, 3]
`pop()` returns 5	[7, 3]

(b)

Solution

Use two stacks: `s1` (inbox) and `s2` (outbox).

Enqueue(x): always push x onto `s1`. Cost $O(1)$.

Dequeue(): if `s2` is empty, pop every element from `s1` and push it onto `s2` (this reverses the order). Then pop from `s2`. If `s2` is not empty, just pop from `s2` directly.

Why this works: moving all elements from `s1` to `s2` reverses their order. The bottom of `s1` (the oldest element, which should dequeue first) becomes the top of `s2`.

Amortized complexity: each element is pushed to `s1` once, moved to `s2` once, and popped from `s2` once — three $O(1)$ operations total per element. Amortized $O(1)$ per enqueue and $O(1)$ per dequeue. Individual dequeue calls can be $O(n)$ (when `s2` must be refilled) but this happens at most once per element.

Solution

(c) Algorithm: scan left to right. For each character:

- Opening bracket (`(`, `[`, {): push onto stack.
- Closing bracket (`)`, `]`, }): if the stack is empty or the top does not match, return false. Otherwise pop.

At the end, the string is balanced iff the stack is empty.

Trace on `"({[()]})"`:

Char	Action	Stack (top on right)
`(`	push	[`(`]
{	push	[`(`, {]
`[`	push	[`(`, {, `[`]
`(`	push	[`(`, {, `[`, `(`]
`)`	top is `(`: match, pop	[`(`, {, `[`]
`]`	top is `[`: match, pop	[`(`, {]
}	top is {: match, pop	[`(`]
`)`	top is `(`: match, pop	[]

Stack is empty at the end: **balanced ✓**

Problem 7 (LeetCode): Two Sum

(a)

Solution

Check every pair (i, j) with $i < j$: if `nums[i] + nums[j] == target`, return $[i, j]$. This uses the array itself implicitly — no auxiliary structure needed.

Time complexity: $O(n^2)$ — there are $\binom{n}{2}$ pairs to check.
Space complexity: $O(1)$.

(b)

Solution

For each element `x` at index `i`: check whether `target - x` is already in the hash table. If so, return the stored index and `i`. If not, store `x` → `i` in the table and continue.

Trace on `[3, 2, 4]`, target = 6:

i	x	target - x	Action
0	3	3	3 not in table; store $3 \to 0$
1	2	4	4 not in table; store $2 \to 1$
2	4	2	2 is in table at index 1; return $[1, 2]$

Time complexity: $O(n)$ — one pass, each lookup and insert is $O(1)$ average.

(c)

Solution

Space complexity: $O(n)$ — in the worst case the hash table stores every element before finding a match (e.g. the answer is the last two elements).

Trade-off: yes, this is a classic time-space trade-off. The naïve solution uses $O(1)$ space but $O(n^2)$ time. The hash table solution uses $O(n)$ extra space to buy down the time to $O(n)$. We spend memory to save time.

Problem 8 (LeetCode): Kth Largest Element

(a)

Solution

Sort the array in descending order and return the element at index $k-1$ (or sort ascending and return index $n-k$).

Time complexity: $O(n \log n)$ for the sort.
Space complexity: $O(1)$ (in-place sort) or $O(n)$ depending on the sort.

(b)

Solution

Maintain a min-heap of size exactly k. The invariant is: the heap contains the k largest elements seen so far, and its root is the smallest of those k (i.e. the k-th largest overall so far).

Algorithm: for each element x:

- If the heap has fewer than k elements, insert x.
- Else if $x >$ heap root (the current k-th largest), pop the root and insert x.
- Otherwise discard x — it is not among the k largest.

At the end, the root of the heap is the k-th largest element.

Why $O(n \log k)$: we process n elements, and each heap operation costs $O(\log k)$ since the heap never exceeds size k.

(c)

Solution

`nums = [3, 2, 1, 5, 6, 4]`, $k = 2$. We maintain a min-heap of size 2.

Element	Action	Heap contents	Heap root
3	heap < 2, insert	[3]	3
2	heap < 2, insert	[2, 3]	2
1	$1 \leq$ root 2, discard	[2, 3]	2
5	$5 >$ root 2, pop 2, insert 5	[3, 5]	3
6	$6 >$ root 3, pop 3, insert 6	[5, 6]	5
4	$4 \leq$ root 5, discard	[5, 6]	5

Answer: root of heap = **5**, which is the 2nd largest element. ✓

(d)

Solution

When $k \ll n$, $\log k \ll \log n$, so $O(n \log k) \ll O(n \log n)$.

Concrete example: suppose $n = 10^6$ and $k = 10$.

- Sorting: $O(n \log n) \approx 10^6 \times 20 = 2 \times 10^7$ operations.
- Heap: $O(n \log k) \approx 10^6 \times \log_2 10 \approx 10^6 \times 3.3 \approx 3.3 \times 10^6$ operations.

The heap approach does roughly $6\times$ less work in this case. As k decreases further, the advantage grows.